The Cray MTA and Unstructured Meshes[*]

Version 1, Sep. 26, 2000

Shahid H. Bokhari
Department of Electrical Engineering
University of Engineering & Technology, Lahore, Pakistan
(shb@acm.org)

Bracy H. Elton
Scientific Libraries Group
Cray Inc.
Seattle, Washington
(elton@cray.com)

Dimitri J. Mavriplis
Institute for Computer Applications in Science & Engineering
NASA Langley Research Center
Hampton, Virginia
(dimitri@icase.edu)

[*]Research supported in part by NASA Contract NAS1-97046 at the Institute for Computer Applications in Science & Engineering, NASA Langley Research Center, Hampton, Virginia. This research was made possible by access to the Cray MTA at the San Diego Supercomputer Center, which receives major support from the National Science Foundation. Additional support was provided by the Optoelectronic Computing Systems Center, University of Colorado, Boulder.

Abstract

The Cray MTA, a multithreaded architecture, is a new parallel supercomputer installed at San Diego Supercomputer Center (SDSC). This machine has an architecture quite different from those of other contemporary parallel machines. It has a flat, shared memory without locality and has hardware support for very fine-grained multithreading. The machine and its associated parallelizing compiler promise great ease in scalable parallel computing.

We report the results of a study, carried out in July–September 1999, to evaluate the execution of EUL3D, a code that solves the Euler equations on an unstructured mesh, on the 8 processor MTA at SDSC. EUL3D captures the essential features of most unstructured mesh codes used in aerodynamic research and development.

Our investigation shows that parallelization of an unstructured code is very straightforward on the MTA. We were able to get an existing parallel code (designed for a shared memory machine), running on the MTA by changing only the compiler directives. Furthermore, a *serial* version of this code was compiled to run in parallel on the MTA by judicious use of directives to invoke the "full/empty" tag bits of the machine to obtain synchronization. This version achieves nearly 250 Mflop/s per processor with little variation as the number of processors is increased from 1 to 8. We achieved this performance without concerning ourselves with the partitioning or placement of data—issues that would be of paramount importance in other parallel architectures.

Our research shows that the fine-grained multithreading possible on the custom built MTA is an interesting alternative to the coarse grained parallelism available on multiprocessors constructed from commodity microcomputers. The recent introduction of commercial microcomputers designed specifically to support multithreading, such as the Intel IXP1200, supports our argument.

Index terms: Aerodynamics, Computer architecture, Euler equations, Parallel computing, Performance evaluation, MTA, Multithreading, Unstructured meshes, Supercomputing.

1 Introduction

The Cray MTA, a multithreaded architecture, is a new parallel supercomputer installed at San Diego Supercomputer Center (SDSC).[1] This machine has an architecture quite different from those of other contemporary parallel machines. It has a flat, shared memory without locality and has hardware support for very fine-grained multithreading. The machine and its associated parallelizing compiler promise great ease in scalable parallel computing.

We report the results of a study, carried out in July–September 1999, in which we evaluated the porting of an unstructured mesh code to the MTA. Algorithms based on unstructured meshes are ordinarily very difficult to parallelize efficiently on conventional parallel machines. Our results show that code can be ported with great ease to the MTA and that the performance achieved is very promising.

We first discuss, in Section 2, how the MTA attempts to compensate for the limitations of conventional parallel machines. We describe the architecture of the machine in some detail in Section 3. In Section 4 we describe our unstructured mesh solver and how it was ported to the MTA. Two variants of the code were ported; the measured performance of these codes is presented in sections 4 and 5. Section 6 presents the results of experiments in which we artificially varied the grain size of the problem in order to make our investigation applicable to a wider range of mesh solvers. In Section 7, we present our conclusions and plans for future research.

2 The State of Parallel Computing

Despite nearly half a century of research and development, truly general purpose parallel computing remains an elusive goal. Very careful programming and a good knowledge of the target computer's architecture are required to achieve even modest performance. At the same time, the wide diversity in available parallel architectures means that a program successfully ported to one machine may require considerable reworking to run well on another. This discourages practitioners from exploiting parallel computing and confines the field to experts, academicians and researchers. Finally, an inordinate effort is required to successfully parallelize an algorithm and, even then, the achieved performance is poor compared with the theoretical peak. There are a number of reasons for this state of affairs.

On currently available distributed memory machines, parallel computing involves a never-ending battle to match computation to architecture. Parallel machines necessarily involve large numbers of interconnected processors. The utilization of these processors is inevitably linked to how well the structure of the computation matches (or can be transformed to match) the structure of the machine. The process of transformation may involve partitioning, mapping and reordering of data, as well as reformulation of the computation. These transformational requirements lead to major combinatorial problems that are often more difficult than the actual problem being solved. The programmer is required to have ex-

[1]The MTA is a product of Cray Inc., which was formerly Tera Computer Company. In April 2000, Tera Computer Company acquired the Cray Research business from Silicon Graphics, Inc. and subsequently changed its name to Cray Inc.

tensive knowledge of the interconnect network, cache hierarchy, arithmetic unit, and other architectural features.

Figure 1 sketches how the quest for utilization has evolved over time on uniprocessors. The checkered rectangles in this figure represent the hardware-time products for the indicated architectures—the higher the utilization, the larger the fraction of grey blocks in this rectangle. A simple, primitive processor's hardware could be utilized only to a limited extent. Among the first developments in computer architecture was the evolution of pipelined processors that could deliver higher utilization for certain types of operations. This higher utilization required additional investments in "performance enhancing" hardware, that is, hardware that did not contribute to actual computation but was required to improve the utilization of the "productive" hardware. A modern pipelined processor improves utilization by considerable investment in such performance enhancing hardware as well as in sophisticated compilers. At the same time, the programmer may have to make some investment in transforming his program, or even the underlying algorithm, to better utilize the specific hardware. Figure 1 also shows that, in a contemporary pipelined machine, some of the work done by the hardware may be wasted because of speculative execution.

A modern parallel processor requires relatively larger hardware and software investments to obtain adequate utilization. Figure 2 illustrates how the productive hardware, i.e., the hardware that carries out the actual computations for our program, has to be augmented with additional hardware and software. A contemporary high performance parallel machine requires performance enhancing hardware in the form of, for example, memory caches, high speed interconnect, synchronization mechanisms, instruction pipelines, and pipelined arithmetic units. Furthermore, considerable investment may be needed in the areas of compilers, operating systems, and analysis tools. Parallel programming platforms such as PVM [5], MPI [7], PARTI [1], PETSc (`www.mcs.anl.gov/petsc`), and OpenMP (`www.openmp.org`) constitute part of the software overhead. The programmer needs to invest considerable effort in developing his program, including rethinking algorithms and, of course, the difficult issues of partitioning, mapping, and scheduling. Despite these overheads, the utilization achieved by such processors is low; indeed, there are large classes of problems for which these machines are considered unsuitable.

One approach to improving utilization is to invest in additional hardware and software to support parallelism, possibly at the expense of additional compiler overhead. Figure 3 illustrates how specially designed hardware can be used to offload the burden placed on the programmer and on parallelism support software. This proposal rules out the possibility of using commodity microprocessors for parallel processing and requires a protracted cycle of development and production. However, the potential benefits are very attractive. The Cray MTA uses this path, as described in Figure 4. By investing heavily in performance enhancing hardware, the MTA is able to eliminate the issues of parallelism support and data partitioning. Higher investment in hardware reduces the effort required by the programmer and also increases the utilization of the productive hardware.

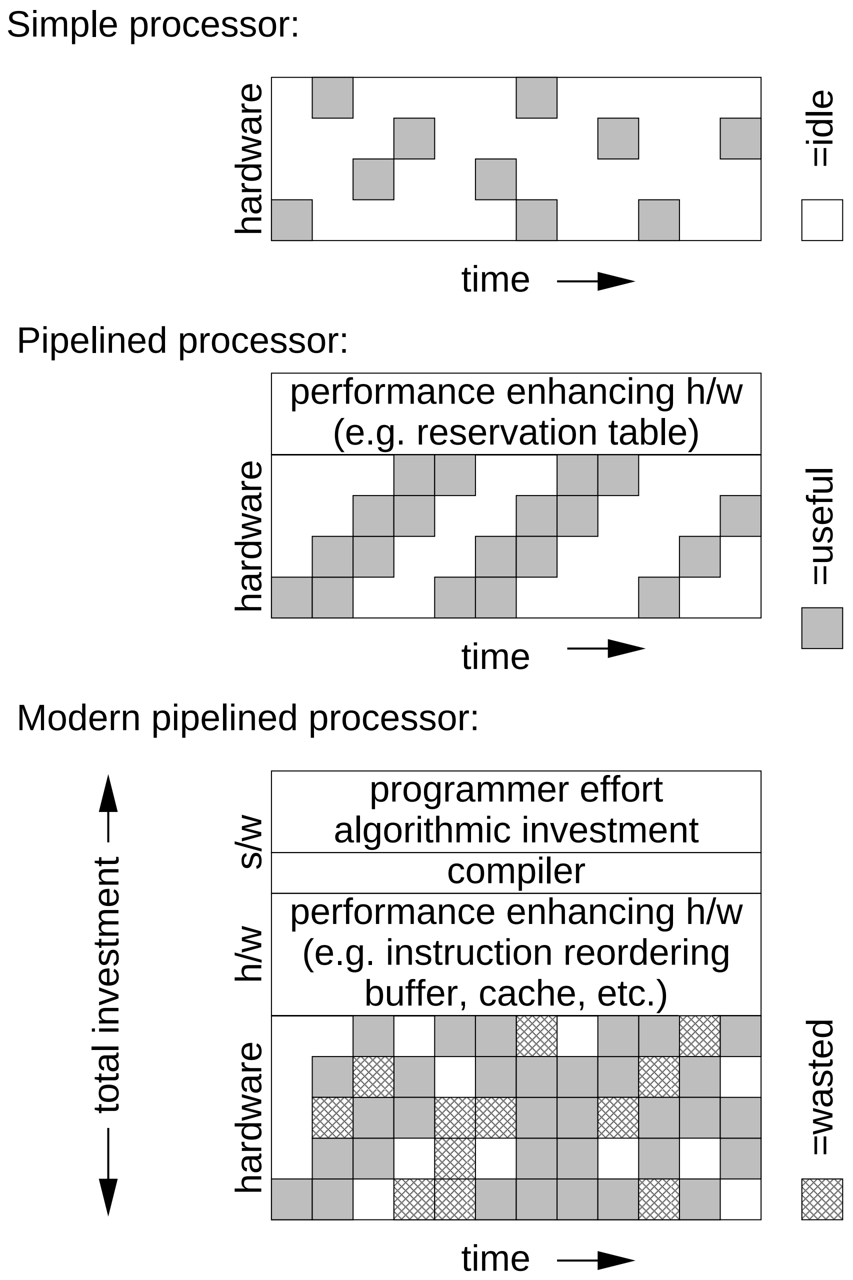
Simple processor:
hardware
time
=idle
Pipelined processor:
performance enhancing h/w (e.g. reservation table)
hardware
time
=useful
Modern pipelined processor:
total investment
s/w
programmer effort algorithmic investment
compiler
h/w
performance enhancing h/w (e.g. instruction reordering buffer, cache, etc.)
hardware
time
=wasted

Figure 1: The Quest for Utilization. As uniprocessors have evolved over time, the investment in non-productive "performance enhancing" hardware has increased. A modern machine also requires considerable investment in compiler development.

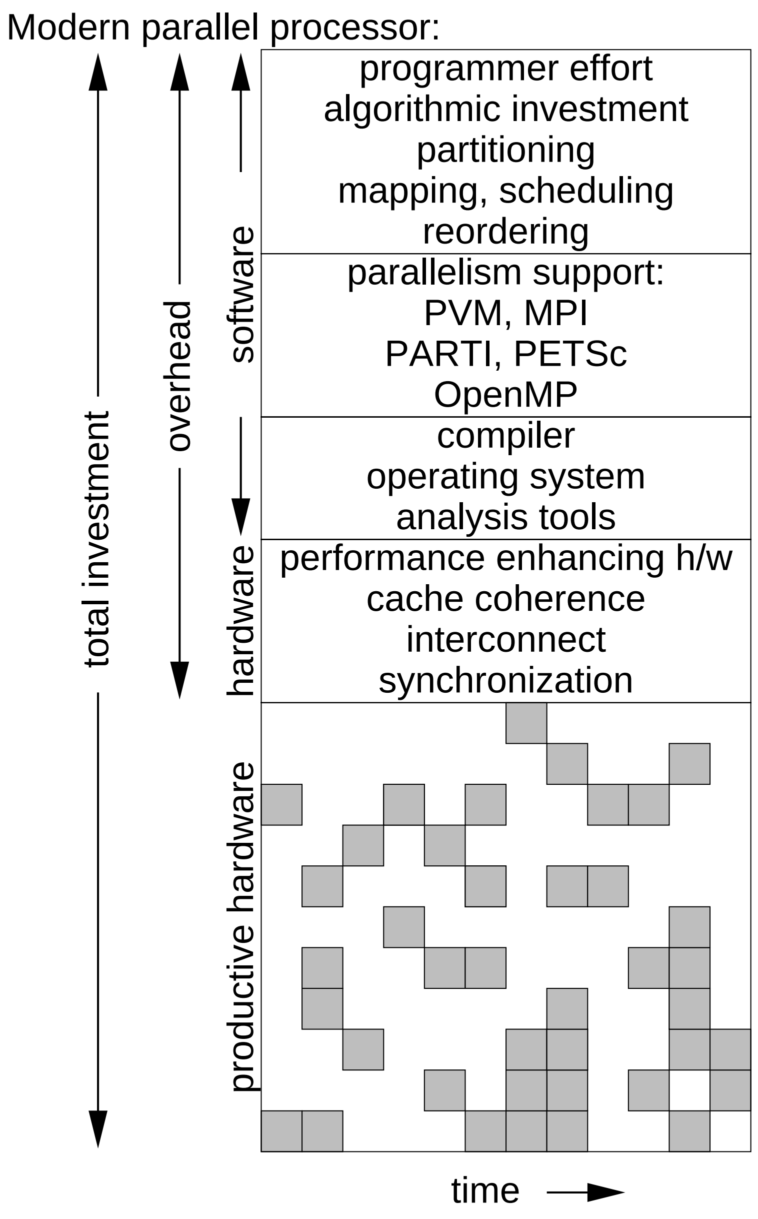

Figure 2: Parallel Computing: Investment and Return. A modern parallel processor achieves low utilization despite considerable investment in hardware and software.

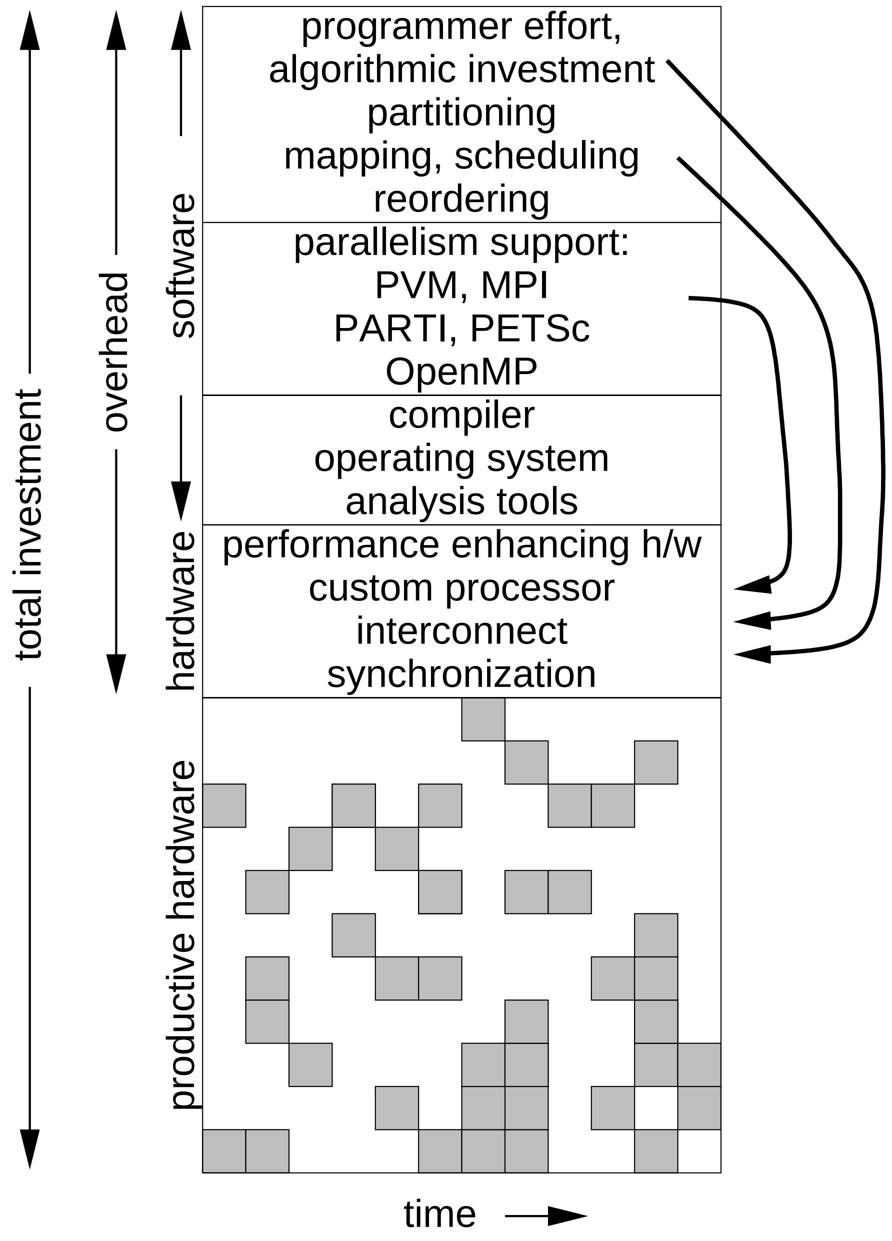

Figure 3: Additional Investment in Hardware Reduces Software Overhead: Functions Migrate into Hardware.

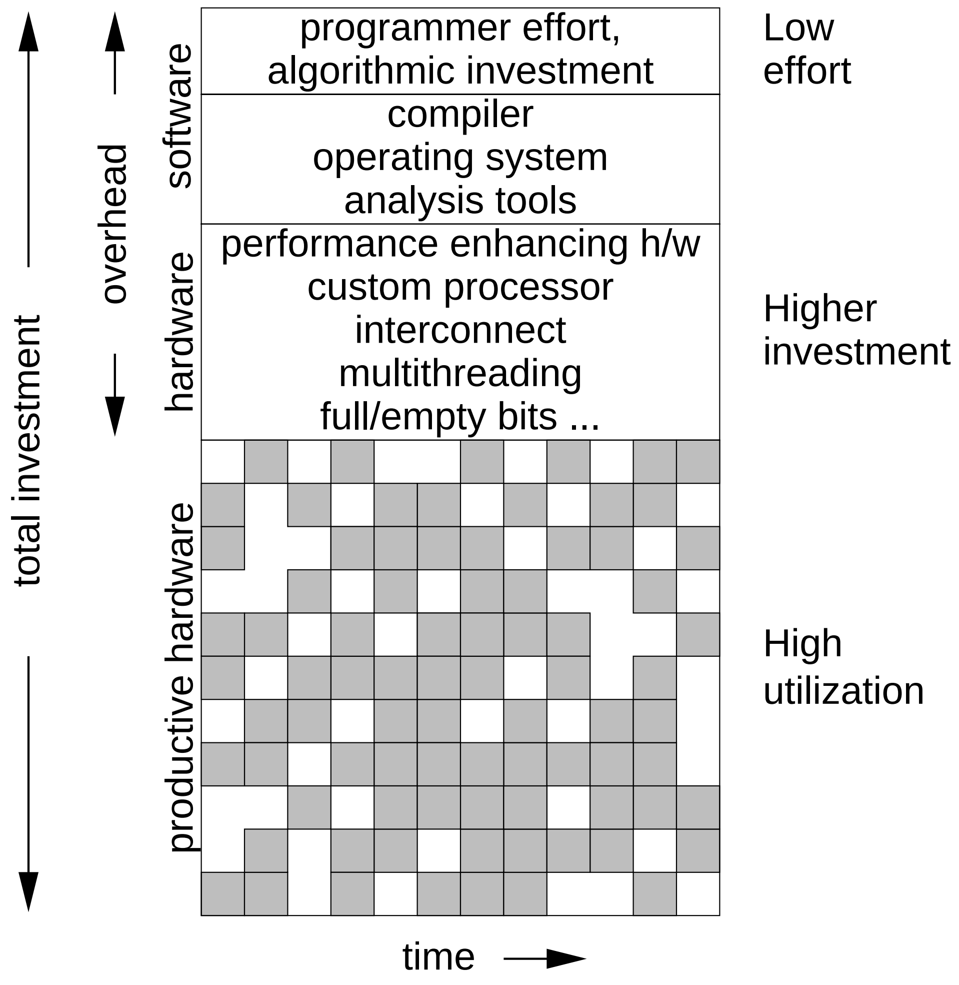

Figure 4: The MTA Idea: Higher Investment in Hardware Yields Improved Utilization and also Reduces Software Overhead.

3 Key Features of the Cray MTA

Detailed information on the architecture may be found in [2, 3]. (The experiences of other researchers in using the MTA are available in [8, 9, 10].) A useful collection of technical papers is available at `www.cray.com/products/systems/craymta/psdocs.html`. We present a brief overview.

3.1 Zero Overhead Thread Switching

The MTA has special purpose hardware (*streams*) that can hold the state of up to 128 threads (per processor). (State includes registers, condition codes, and a program counter.) On each clock cycle, each processor switches to a different resident thread and issues one instruction from that thread. A blocked thread, e.g., one waiting for a word from memory or for a synchronization event, generally causes no overhead—the processor just executes the instructions of some other ready threads.

3.2 Pipelined Processors

Each processor in the MTA has 21 stages. As each processor accepts an instruction from a different stream at each clock tick, at least 21 ready threads are required to keep it fully utilized. Since the state of up to 128 streams is kept in hardware, this target of 21 ready threads is easy to achieve.

3.3 Flat Shared Memory

The MTA has a byte addressable memory. Full/empty tag bits (described below) are associated with 64 bit words. Addresses are scrambled by hardware to scatter them across memory banks [2, 4]. As a result, the memory has no locality, and there are no issues of partitioning data or mapping memory on the machine.

Regarding latencies, the cycle time per memory bank is 35–40 clock ticks, depending on the system's clock frequency. (At 255 MHz, as is the case for our experiments, this latency is about 38 ticks.) The access time varies from 150–200 clock ticks, depending upon the size of the system. The 21 stage processor pipeline is dwarfed by the some 150 cycles of latency to memory. These latencies are overcome by having programs run with *more* than 21 threads. Additionally, by using memory lookahead each thread can have multiple memory references outstanding. While it is guaranteed that arithmetic operations in one instruction are completed before others in the next instruction in the same thread, memory operations can be issued up to seven instructions before they are needed to be completed. Consequently, using this lookahead feature a thread can have up to eight memory references outstanding at any given instance. A processor typically has hundreds of memory references outstanding.

3.4 Extremely Fine-grained Synchronization

Each 64 bit word of memory has an associated *full/empty* bit. A memory location can be written into or read out of using ordinary loads and stores, as in conventional machines.

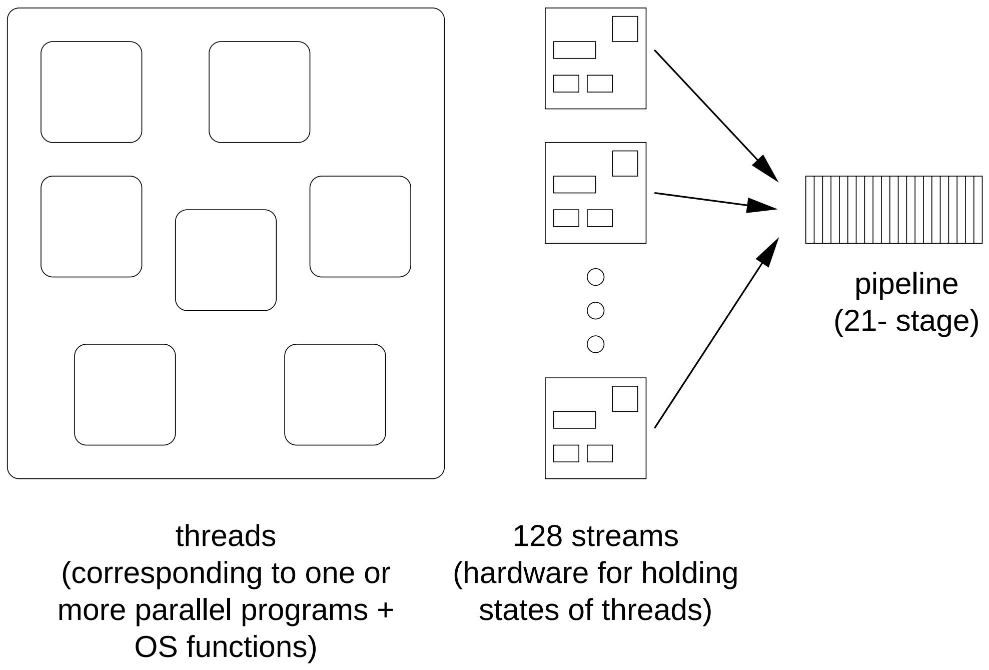
pipeline
(21- stage)
threads
(corresponding to one or more parallel programs + OS functions)
128 streams
(hardware for holding states of threads)

Figure 5: The MTA (1 processor).

Load and store operations can also be under the control of the full/empty bit. For example, a "read-when-full, then set-empty" (`y=readfe(x)`) operation atomically reads data from a location only after that location's full/empty bit is set *full*. The full/empty bit is set *empty* during the read operation atomically with reading the data. If the full/empty bit is not set, the thread executing the read operation suspends (in hardware) and is later retried. The thread resumes when the read operation has completed. This is a low overhead operation since the thread is simply removed from and later reinserted into the ready queue. This feature allows extremely fine-grained synchronization and is detailed in Section 5.1.

3.5 An Analogy

The aspects of the MTA discussed above give it great flexibility in attacking parallel problems. This is best explained with an analogy. Parallel processing may be viewed as the process of harvesting a wheat field. A large, conventional parallel processor is equivalent to a team of combine harvesters sweeping through a large field. Such a team works best when the field is large, uniform and rectangular in shape. Should the field be irregularly shaped and have variations in density, the team will lose efficiency and perform poorly.

The MTA, in contrast, may be viewed as a huge swarm of insects picking up individual grains. Such a swarm is insensitive to the shape of the field or to any variations in density. The challenge for each member of the swarm is to transport the grains very quickly and to move to the next available grain without significant delay. The custom designed hardware of

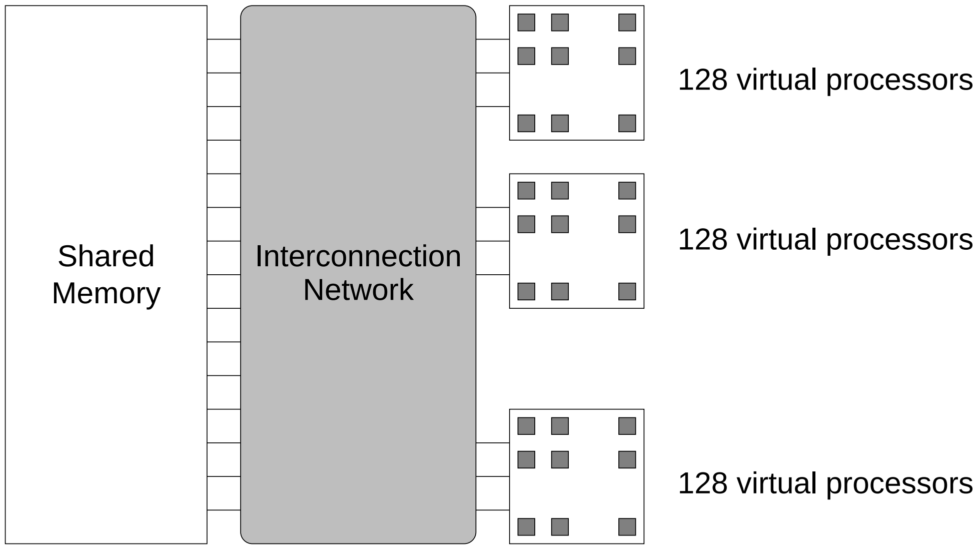

Figure 6: A View of the MTA Processor. Each stream may be thought of as a virtual processor. Some streams may be needed to execute OS functions—a user may not be able to use all 128 streams per processor.

the MTA satisfies these requirements. In effect, the hardware of the machine is partitioned into small, agile and autonomous units. This, in turn, relieves the programmer from the onerous task of partitioning his problem. Thus, the one time development cost of partitioning the hardware eliminates the recurring cost of partitioning problems.

Parallel machines made up of conventional commodity microprocessors cannot be made to behave in this fashion. Interestingly, recent announcements from industry describe microprocessors with some support for multithreading. The Intel IXP1200 [6] is one such example, though its target market is telecommunications and not parallel scientific computing.

3.6 MTA Performance Characteristics

At the time of our experiments (July–September, 1999), the clock of the MTA was running at 255 MHz. There are three units in each processor, all of which may be active during a single cycle:

Unit	Operation	Max flop
M (Memory)	–	0
A (Arithmetic)	fused multiply-add	2
C (Control)	add	1
	Total	3

Thus "peak" performance is $3 \times 255 = 765$ Mflop/s. On one processor of the MTA, the unstructured mesh code ran at about 250 Mflop/s with a system clock frequency of 255 MHz, seeing just about one floating point operation per cycle time. We discuss more about our experiments later, where our code achieved almost 2 Gflop/s on 8 processors.

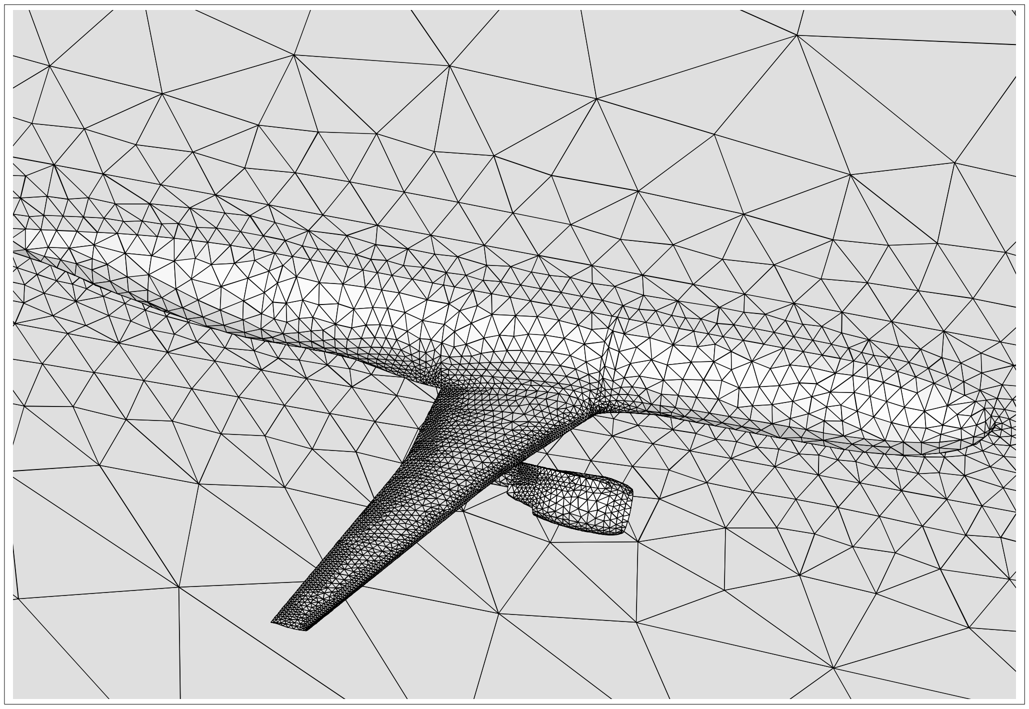

Figure 7: Unstructured Meshes are Widely Used in Aerodynamic and Structural Analysis Codes. Because of the enormous, irregular variations in density, algorithms based on such meshes are difficult to parallelize on conventional multiprocessors.

4 The Numerical Solver

The code that we chose to implement on the MTA is a representative kernel from EUL3D, a 3D unstructured grid Euler solver. This code uses vertex based variables and an edge-based loop for residual construction. The kernel reproduces edge-based flux loops and vertex based updates.

Unstructured mesh problems have traditionally been difficult to parallelize because of their need for partitioning, mapping and load balancing. Furthermore, because of the indirect access to the grid data, such problems are hard to compile.

On the MTA these become non-issues because

1. Data partitioning and mapping are not needed with the flat shared memory which has no locality, and

2. Explicit load balancing is not needed because of very fine-grained multithreading and an intelligent compiler: loops can be dynamically scheduled across processors with very little overhead.

The specific problem with which we experimented has 53,961 nodes and 353,476 edges. This is considered to be a medium-sized problem in the aerodynamics community—a large

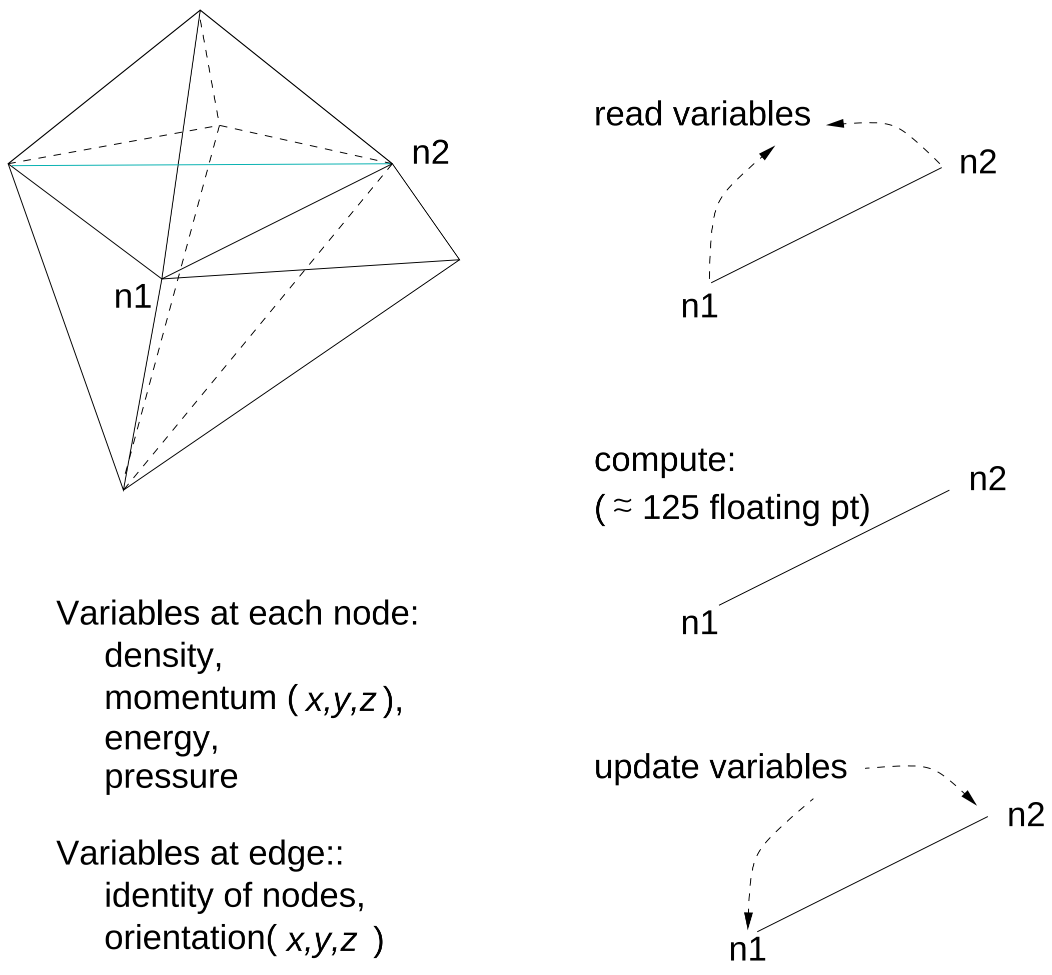

Figure 8: Computation in the Edge-Based Loop

problem would have 0.3 million nodes and 3 million edges. We anticipate that a larger problem would scale better than the one we studied.

At each node of our mesh we store density, momentum (x, y, z components), energy, pressure, plus some scratch space. This results in approximately 10 variables per node.

For each edge we need to store the identity of the 2 nodes at its end points plus a vector describing the orientation of the edge. We thus have about 5 variables per edge.

The movement of data in the edge-based loop is described in Figure 8. Pseudocode corresponding to this loop is given below.

```
do i=1, totalNodes
  initialize variables
enddo
```

```
do cycle=1, totalCycles
   do i=1, totalNodes
     clear residuals
   enddo

   do i=1, totalEdges
     compute residuals
   enddo

   do i=1, totalNodes
     update variables
   enddo
enddo
```

4.1 Parallel Implementation

When executing the edge-based loop in parallel, it is important to ensure that two threads do not attempt to update the same node at the same time. One way of ensuring this is to color the edges of the graph so that no edges incident on same node have the same color. Once this has been done, all edges with the same color can be processed in parallel.

Although the problem of finding the minimum color edge coloring of a graph is intractable, our primary objective is to obtain a coloring with a reasonable number of colors. A simple greedy algorithm is fast and effective for our purposes. On our sample problem, which has average degree 14, the algorithm yields 24 colors.

In the pseudocode for the edge-colored algorithm, given below, the compiler has to be told to parallelize the edge loop. This is because it has no way of knowing about the coloring, and cannot establish that it was safe to parallelize the loop just by looking at the code. The `C$TERA ASSERT PARALLEL` compiler directive is used for this purpose.

```
do i=1, totalNodes
  initialize variables
enddo

do cycle=1, totalCycles
   do i=1, totalNodes
     clear residuals
   enddo

   do i=1, totalColors
C$TERA ASSERT PARALLEL
      do (for each edge of color i)
        compute residuals
      enddo
   enddo
```

```
          do i=1, totalNodes
            update variables
          enddo
        enddo
```

4.2 Performance of Edge-colored Algorithm

The performance of the edge-colored algorithm was measured by varying the number of streams (1 to 128), and processors (1 to 8). The MTA compiler normally selects the number of streams for each parallel loop, based on estimated grain size and expected number of iterations. Instrumentation was developed to request a specific number of streams and report the number of streams actually allocated. All requested streams may not be allocated, even if a program is run in standalone mode, since some, e.g., may be required by the operating system. When interpreting run-time data, it is important to ensure that that the number of streams granted matches the number requested. We measured the performance of EUL3D under almost standalone conditions and were able to get up to 80–90 streams per processor. The performance of EUL3D saturates at 60-70 streams per processor, so we see little impact when the number of streams granted is less than the number of streams requested, provided we get at least 80 streams.

The curve in Figure 9 shows the performance of the edge-colored algorithm as the number of streams is varied. The plot labeled `1 processor` shows the performance of the algorithm on one processor. The time per cycle drops very smoothly from 1 to 30 streams and flattens out at 60 streams. In the present context, we define speedup as follows.

$$\text{speedup} = \frac{\text{time to execute the algorithm with one stream}}{\text{time to execute the algorithm with } n \text{ streams}}.$$

The straight line next to this curve shows the time to run the algorithm under ideal speedup, i.e., time to execute the algorithm with one stream divided by the number of streams. From the figure we can see that the speedup with one processor is about 60, since the curve for one processor flattens off at a level that would ideally be achieved by 60 streams. In this case there would be no advantage to using more than 60 streams with one processor. The decision on the number of streams to use per processor is normally made by the compiler and is not under user control. For our experiments, we developed special routines to control the number of streams under program contol.

The curve labeled 2 shows the performance of this algorithm on two processors. This curve plots the performance of the algorithm for $2, 4, 6, \ldots, 256$ streams (which correspond to $1, 2, 3, \ldots, 128$ streams *per processor*). In general, the ith curve plots the performance for $i, 2i, 3i, \ldots, 128i$ streams on i processors. This depiction lets us compare the performance of the algorithm on a varying number of processors with a simple 2-dimensional plot.

We can see that the performance of the algorithm improves smoothly as the number of processors is increased, with some indication of saturation going from 7 to 8 processors. We conjecture that this is due to interference from operating system activities that must necessarily run on the the last processor.

We have marked a scale indicating the performance (in Mflop/s) achieved on the right hand side of this plot which shows that on 8 processors the program achieves 1.7 Gflop/s.

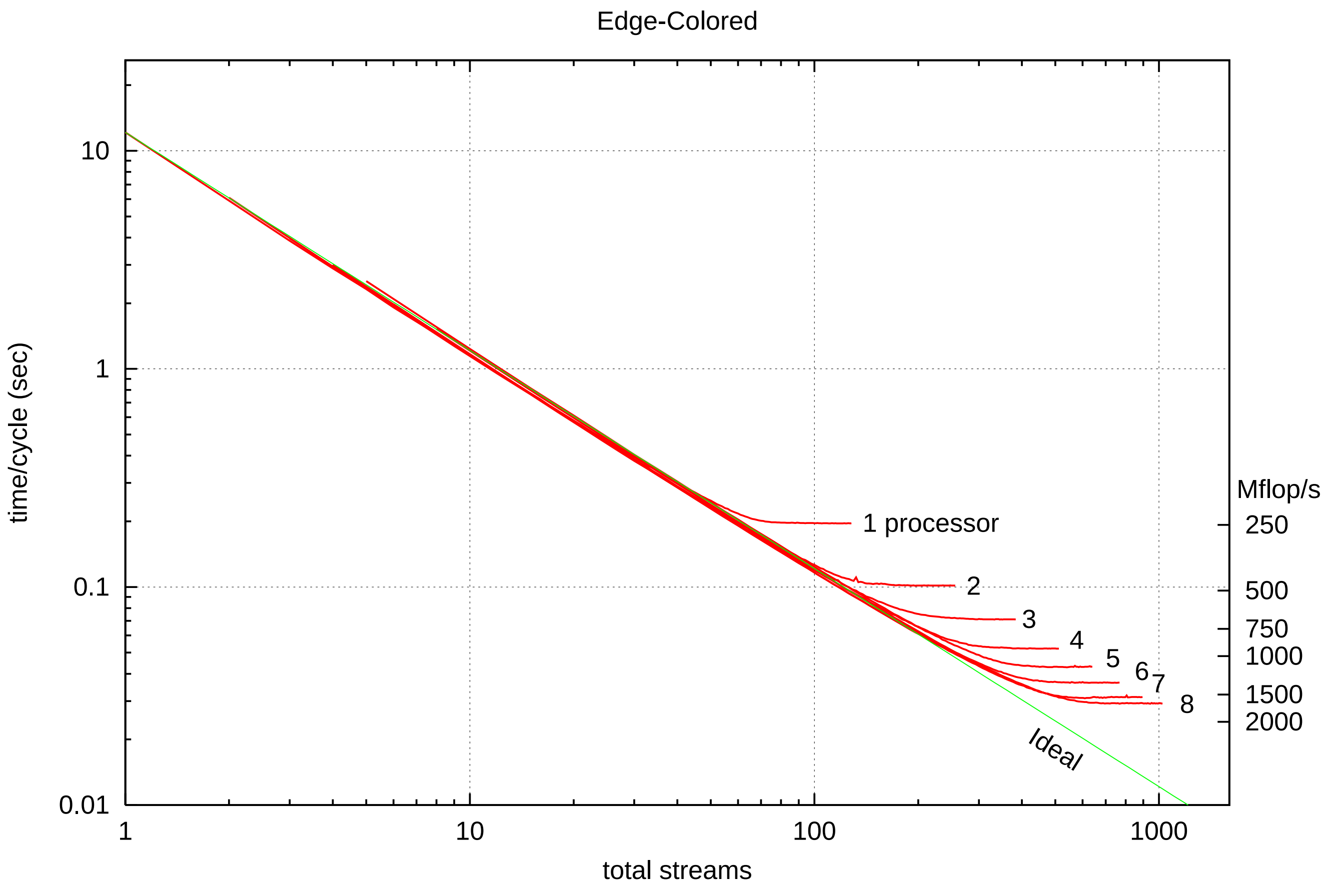

Figure 9: Performance of Edge-Colored code. The curve labeled "1 processor" represents the times for $1, 2, 3, \ldots, 128$ streams (on 1 processor.) The one labeled "2" shows the times for $2, 4, 6, \ldots, 256$ streams (on 2 processors), and so on.

5 The Synchronized Algorithm

The edge-colored algorithm presented above has two overheads:

1. The time required to actually color the edges and reorganize data (this is a one time cost, assuming the mesh is static), and

2. The overhead of executing the color loop (this includes synchronization overhead at the bottom of the loop).

In the edge-colored algorithm, we colored the edges and processed only like-colored edges in parallel to avoid race conditions in updating variables. These updates appear in the form, such as,

```
dw(i) = dw(i) - xincr
```

The full/empty bits of the MTA permit very fine-grained synchronization, and thus updates to variables such as `dw` above can be done atomically. This lets us eliminate the overheads associated with edge-coloring. Furthermore, because updates are done atomically for individual update statements, there is less contention than in the case where an entire section of code is marked as a critical section. Consequently, using the synchronization feature allows the code to scale to larger numbers of threads. The *serial* algorithm can be run in parallel on the MTA, provided the compiler is warned, if it does not determine this on its own, about the sections of codes where it should ensure atomic updates. In this case, the preprocessing step of coloring and reorganizing data is not required and the overhead of the color loop and its associated synchronization costs are avoided.

5.1 Using the Full/Empty bits

The behavior of the MTA's full/empty bits may be summarized below. Note that the operations described are atomic with respect to reading or writing and changing the state of the full/empty bit.

- A synchronized write into a variable succeeds when it is *empty*. If the variable is *full* then, the write blocks until it becomes *empty*. When the write completes, the location is set *full*.

  So, a thread attempting a synchronized write into a *full* location will be suspended (by *hardware*) and will resume only when that location becomes *empty*.

- A synchronized read from a variable succeeds when it is *full*. If it is *empty*, then the read blocks until it becomes *full*. When the read completes, the location is set *empty*.

  So, a thread attempting a synchronized read from an *empty* location will be suspended (by *hardware*) and will resume only when that location becomes *full*.

There are several ways of using the full/empty bits, as detailed below. They are quite effective at atomically updating variables, such as those that appear in the unstructured mesh code. For example, in the code

```
dw(i) = dw(i) - xincr
```

the update to `dw(i)` can be done as follows:

1. Perform a synchronized read of `dw(i)`.
2. Perform the subtraction (in registers).
3. Store the result to `dw(i)` under a synchronized write.

This way the update to `dw(i)` is guaranteed to be atomic with respect to other loop instantiations wanting to update the same `dw(i)`.

5.1.1 Synchronized Variables

In Fortran a variable can be declared synchronized thus:

```
      sync real dw(100)
```

or, via directive:

```
C$TERA SYNC dw
      real dw(100)
```

In this case, writes and reads to/from `dw()` will follow the full/empty rules given above. This approach requires careful thought and is not recommended for porting existing codes. However, it may result in concise and elegant code when a program is written from the ground up with synchronized variables in mind.

5.1.2 Machine Generics

Machine language instructions such as `writeef()` ("wait until a variable is empty, then write a value into it and set the full/empty bit to full") can be invoked from within Fortran or C. These are known as *machine generics.*

To ensure that the Fortran update

```
      dw(i) = dw(i) - xincr
```

is handled properly when several threads are using the same value of `i`, we could use

```
      call writeef(dw(i), readfe(dw(i)) - xincr)
```

Machine generics such as `writeef` and `readfe` are not compiled into function or subroutine calls—they become *individual* MTA machine instructions.

This technique is the most flexible and gives full control to the programmer. He or she has the option of using regular or synchronized (a la the full/empty bit) load /store operations on any particular variable. One advantage is that a single variable can serve to lock a whole code section, which may be desirable to implement a large critical section. On the other hand, this must be balanced with the rest of the computation. If the section is too large relative to the rest of the computation, then it can increase the likelihood of contention.

A disadvantage in this approach is that extensive use of `writeef`, `readfe` and other generics can obscure the code and makes it difficult to read.

5.1.3 Compiler Directives

Compiler directives can be used to make the compiler use full/empty bits to ensure correct updating. For example, in the following code fragment,

```
C$TERA UPDATE
      dw(i) = dw(i) - xincr
```

the directive instructs the compiler to insert appropriate machine instructions to insure that the update to `dw(i)` is atomic.

This is the cleanest solution as it requires no change to serial code and does not obfuscate the program text. This is the solution we have used. However, this approach may not work in all situations.

5.1.4 Compiler Detection

It is also possible for the compiler to detect program statements where use of full/empty bits would be required and insert the required machine instructions. This is the least intrusive solution but, as in the synchronized approach described above, may not work in all cases. At the time of our experiments, the compiler could not automatically parallelize the update loop and ensure atomic updates are used. Today, the compiler can parallelize the loop and do atomic updates without any directives.

5.2 Performance of the Synchronized Code

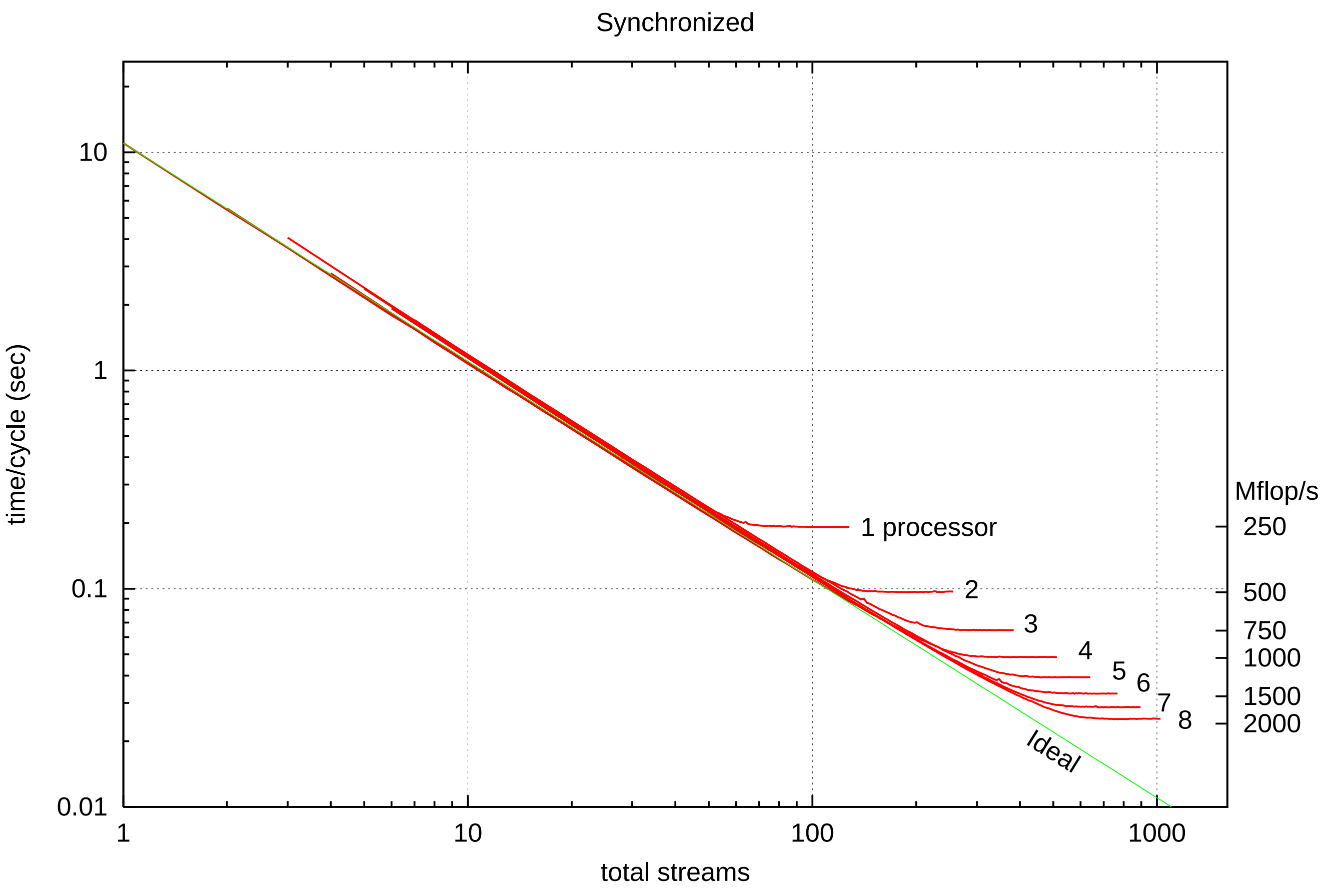

Figure 10: Performance of Synchronized Code.

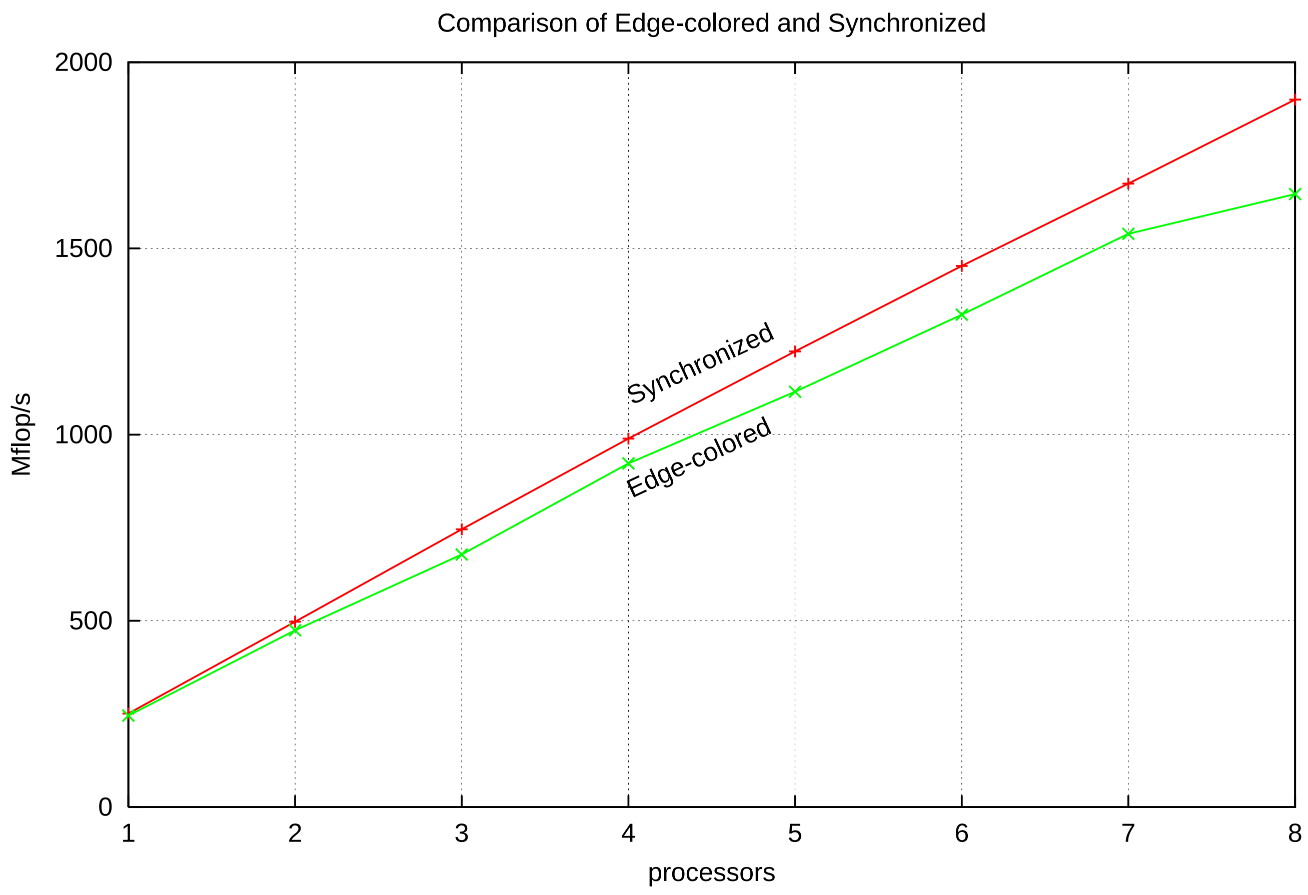

Figure 11: Comparison of Synchronized Code with Edge-Colored Code.

The performance of the synchronized code is shown in Figure 10. A comparison of the original synchronized code and the edge-colored code is given in Figure 11. It is evident from these figures that the synchronized code is significantly better than the edge-colored code. Its performance is almost linear with increasing processors, and it achieves nearly 2 Gflop/s on 8 processors.

Recall that the synchronized code is just the *serial* code with the addition of a few compiler directives. These directives cause the MTA to use its full/empty bits to ensure correct updating. This eliminates the overhead of the edge color loop and its associated synchronization. These directives were necessary for correct parallelization at the time our experiments were done. The current version of the MTA compiler does not require the use of any directives.

It is interesting to note that the edge-colored code has a significant drop-off at 8 processors. The overheads in this code presumably saturate the machine more severely than is the case with the synchronized code.

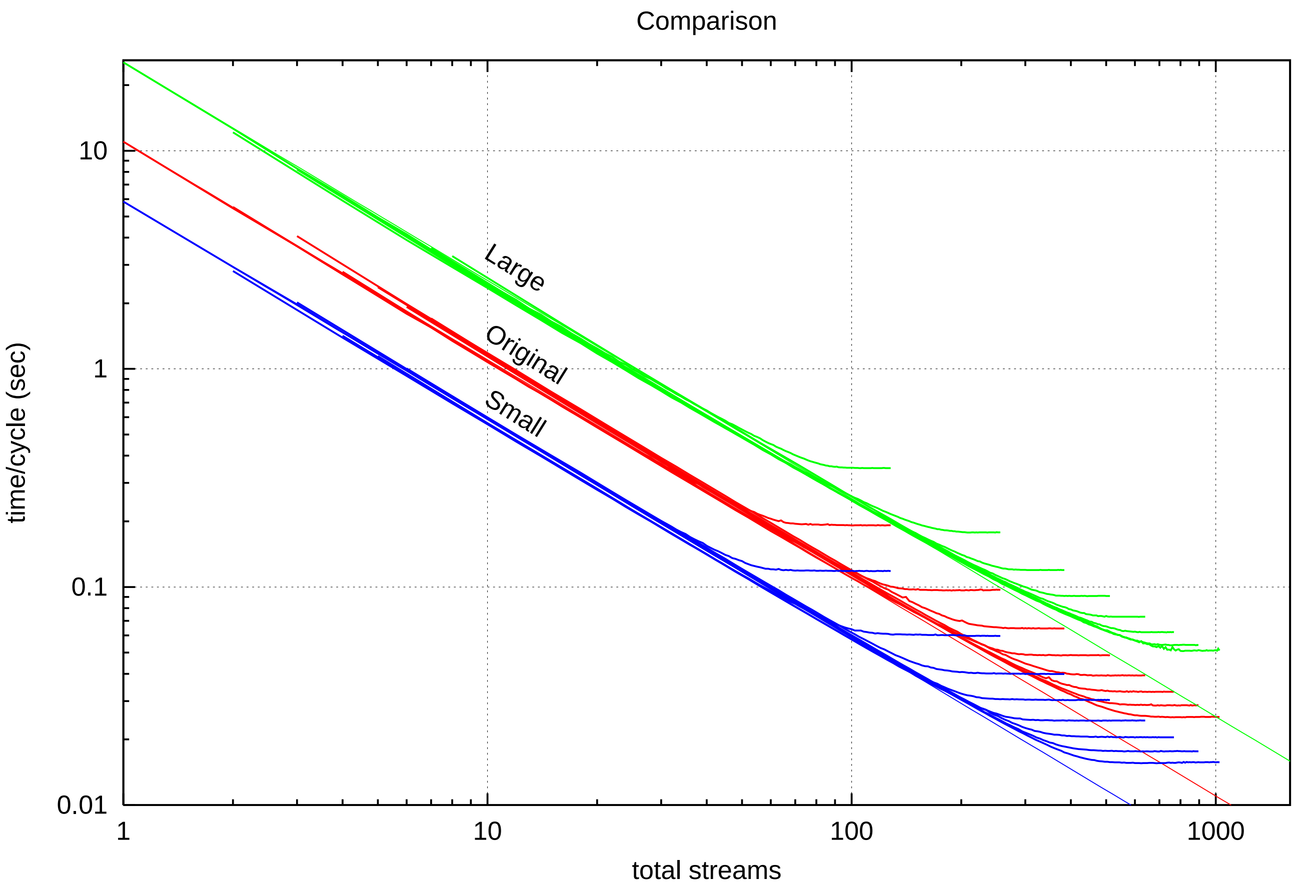

Figure 12: Effect of Varying Grain on Run Time.

6 Experiments with Grain Size

The "edge-based" loop in EUL3D is used in numerous other unstructured mesh problems. Other problems might have grain sizes very different from EUL3D. To get an idea of how performance varies with changing grain size, we artificially modified the synchronized version of the EUL3D solver, which was described in section 5 above. In the remainder of this paper, we refer to this synchronized version as the "original" code. The experiments in this section were suggested by David Keyes of ICASE. Harry Jordan of the University of Colorado assisted us with the analysis that follows.

The original solver has 12 variables per node. We modified these to 6 and 22. We also modified the computations in the edge loop to roughly halve or double them.

The results of these experiments are summarized in Figure 12. The plots show that the speedup curves follow generally the same pattern. In the small code, performance saturates somewhat earlier (i.e., for a smaller number of streams) than the original code, which in turn saturates before the large code. This is because there is more work available per iteration as we increase the grain size. The large code's performance envelope shows a rounder knee, indicating that its performance is saturating. This is confirmed by Figure 13 which shows a distinct drop-off in Mflop/s for the 8th processor. At the time our experiments were done,

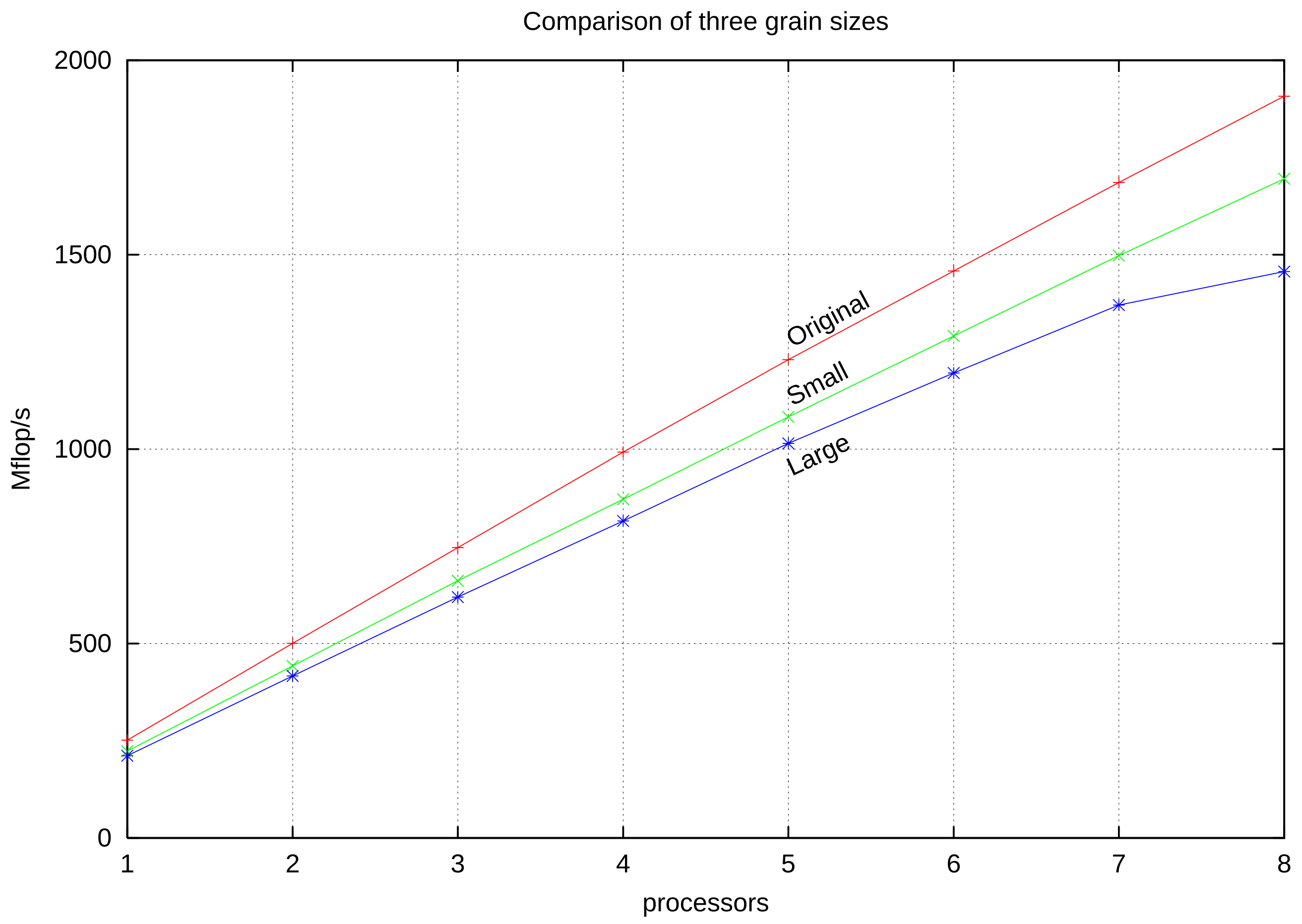

Figure 13: Effect of Varying Grain on Performance in Mflop/s.

the SDSC MTA had more disabled links than the design anticipated in the processor-memory network. Consequently, performance on the maximum number of processors in a system (8 in our case) were somewhat degraded.

To analyze these results in detail we turned to the CANAL (Compiler Analysis) tool that accompanies the MTA compiler. This tool supplied us with the instruction counts for the various loops of the three codes as well as the expected number of repetitions of these loops (in our case these were the numbers of nodes and edges in the mesh). In addition to instruction count, CANAL indicates the number of floating point operations in each loop.

This information is summarized in Table 1. This table enumerates the predicted and observed performance on one processor for the three grain sizes. The entries in this table are, for each grain size, the number of instructions executed in one program cycle (that is, the cycle whose timings are given in Figure 12). The table also lists the number of floating point operations (flop) in each cycle. The flop count can exceed the number of instructions because the MTA is capable of launching more than one floating point operation per instruction. By dividing this by the number of instructions we can obtain the predicted performance in units of flop/tick. Since the machine was operating at 255 MHz at the time these experiments were done, multiplying flop per clock tick by 255 gives us the predicted performance in Mflop/s.

Table 1: Predicted and measured performance for various grain sizes on one processor.

Grain	Instructions	flop	Mflop/s		
			Predicted	Measured	Ratio
Small	27,218,171	26,356,691	246.93	223.13	0.90
Original	46,432,589	48,153,363	264.45	251.58	0.95
Large	82,735,110	74,043,197	228.21	211.22	0.93

The measured performance in Mflop/s is obtained by dividing the flop count by the actual time needed to complete one cycle. There is close (but not perfect) agreement between the predicted and measured performance (Mflop/s). This mismatch is explained by the fact that our analysis, based on CANAL output, does not include the overhead for starting up the parallel regions, loop control, and other compiler-controlled bookkeeping mechanisms. Furthermore, CANAL includes information only for loops and thus its output and the actual instruction count differ somewhat.

Of greatest interest is the fact that the lower measured performance of the large grained code is accurately predicted by our analysis. The large grained code's main loop has more work. This requires more registers than are available. The compiler ends up having to generate code to save and reload the registers to/from memory. Consequently, the efficiency of the loop, relative to the number of instructions needed to implement it, is lower than that for the loops with less work per loop body. Nevertheless, it scales well on 1–7 processors.

Table 2 lists the measured speedup of the three grain sizes. For this table, speedup is defined as

$$\text{speedup} = \frac{\text{best time to execute the algorithm on one processor}}{\text{best time to execute the algorithm on } n \text{ processors}},$$

where "best" means the minimum over all possible numbers of streams. This minimum corresponds to the flattened tails of the curves in Figure 12.

For the small and original grain sizes we obtain a speedup of 7.6 for 8 processors which is 95% of ideal. The large grained code has poorer performance with a speedup of 6.9, corresponding to 86% of ideal. This is conjectured to be the result of the much heavier memory access requirements of this code while running with a somewhat handicapped processor-memory network, the handicap affecting mostly memory access of the 8th processor.

7 Conclusions

Our experience with the Cray MTA has generally been positive. We were able to port an existing edge-colored parallel code (previously run on the SPP-2000 at Caltech) by changing only the parallelization directives.

We also parallelized an existing serial code, which is the workstation version, on the MTA with the addition of a few compiler directives. In this case we invoked, through the

Table 2: Measured speedup for various grain sizes on 1–8 processors.

Procs.	Small		Original		Large	
	Mflop/s	Speedup	Mflop/s	Speedup	Mflop/s	Speedup
1	223.13	1.00	251.58	1.00	211.22	1.00
2	442.32	1.98	500.28	1.99	416.57	1.97
3	661.20	2.96	746.70	2.97	619.62	2.93
4	870.98	3.90	992.55	3.95	815.37	3.86
5	1082.81	4.85	1229.91	4.89	1014.89	4.80
6	1290.92	5.79	1457.87	5.79	1195.85	5.66
7	1497.71	6.71	1685.75	6.70	1370.41	6.49
8	1695.07	7.60	1907.59	7.58	1456.74	6.90

use of full/empty bits, the word-based synchronization mechanism of the machine and thus eliminated the overhead of the edge-colored loop. A more recent version of the compiler is capable of parallelizing this code without any directives.

Both versions of our code were run on 1 to 8 processors. No changes will be required to run on any additional processors.

The edge-based loop used in EUL3D is at the heart of many other unstructured mesh algorithms. It will therefore be of interest to port other unstructured mesh problems to the MTA.

Cell-based (as opposed to edge-based) loops should be similarly easy to parallelize and need to be investigated. Finally, we plan to port other non-uniform problems, such as multiblock. The MTA's insensitivity to memory access patterns, ability to tolerate memory latencies via low-level multithreading, and support for synchronized memory access will be major assets for such problems.

8 Acknowledgments

We are grateful to Manuel Salas, Director ICASE, for his encouragement of this research and to Wayne Pfeiffer for arranging access to the Cray MTA at San Diego Supercomputer Center. John van Rosendale, David Keyes, Piyush Mehrotra, Tom Crockett and Harry Jordan provided valuable assistance. Allan Snavely, Simon Kahan, David Callahan, Andy Kopser, Brian Koblenz and John Feo generously shared their knowledge of the MTA with us and lightened our burden considerably.

This research would not have been possible without the hospitality of David Chestnut, Oktay Baysal and Richard Barnwell at the Virginia Consortium of Science & Engineering Universities (VCES).

Additional support was provided by Jon Sauer at the Optoelectronic Computing Systems Center of the University of Colorado at Boulder.

9 Web Sites of Interest

`www.cray.com/products/systems/craymta/`
`www.npaci.edu/MTA`
`www.sdsc.edu`
`www.icase.edu`

References

[1] D. J. Mavriplis, R. Das, J. Saltz and R. E. Vermeland. Implementation of a parallel unstructured Euler solver on shared and distributed memory machines. *The Journal of Supercomputing.* 8(4):329-344, 1995.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield and B. Smith. The Tera Computer System. In *1990 International Conference on Supercomputing.* pp. 1–6, June 1990. Also, appears in *ACM SIGARCH Computer Architecture News.* 18(3), Sept. 1990.

[3] G. Alverson, R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. Smith. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In *the 1992 International Conference on Supercomputing.* pp. 188–197, July 1992.

[4] A. Norton and E. Melton. A class of boolean linear transformations for conflict-free power-of-two stride access. In *Proceedings of the 1987 International Conference on Parallel Processing.* 247–254, August 1987.

[5] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam. *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press. 1994.

[6] M. Levy. Development tools unleash network processors' power. *Electronic Design News,* p. 115 February 3, 2000. `www.ednmag.com/ednmag/reg/2000/02032000/03tt.htm`. Current as of September 1, 2000.

[7] W. Gropp, E. Lusk and A. Skjellum, *Using MPI.* MIT Press. 1994.

[8] A. Snavely, L. Carter, J. Boisseau, A. Majumdar, K. S. Gatlin, N. Mitchell, J. Feo and B. Koblenz. Multi-processor Performance on the Tera MTA. *Supercomputing 98,* November, 1998. `www.sdsc.edu/~allans/SC98-MTA/abstract.html`. Current as of September 1, 2000.

[9] L. Carter, J. Feo and A. Snavely. Performance and programming experience on the Tera MTA. *SIAM 99,* March 1999. `www.cs.ucsd.edu/users/carter/Papers/siam99.ps`. Current as of September 1, 2000.

[10] L. Oliker and R. Biswas. Parallelization of a Dynamic Unstructured Application using Three Leading Paradigms. *Supercomputing 99,* 1999. `www.nersc.go /~oliker/papers/sc99.pdf`. Current as of September 1, 2000.